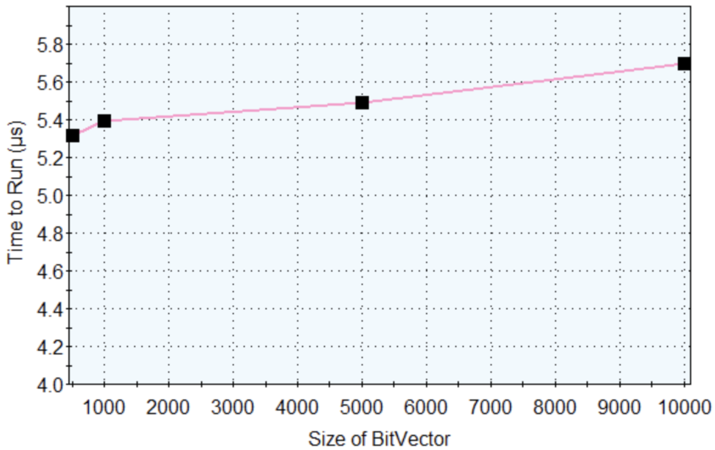
Task 1

To create the RankSupport structure I first need to gain access to the bitvector, this could be loaded in using the load bit vector function. This bitvector would be passed into a function that loops through it and creates two different vectors. The first vector is the larger chunks used in the implementation of rank support and the second vector is the smaller chunks of the larger chunks. Each of these vectors stores the exclusive rank for the start of the chunk. Along with these two structures, a lookup table was required to store the rank of each index, dependent on the small chunk it was located in. These, along with the bitvector, are added to a new RankSupport structure. From this point functions like rank1, overhead, save, and load can be run on the Rank Support struct. Rank1 needed to do the specific math that combined all information from the created structures listed above to discover the rank. The general idea was to find the index in the chunk that is being focused on and take that rank adding it with the other two ranks from the other two structures. Added together they provide the rank at a specific index.

The part of this task that I struggled the most with was doing the population count in the rank1 function. While a lot of languages have a population count operation, Rust does not come with one already built in nor was there one in the BitVec implementation chosen. This meant I had to find another way to do the final step of the rank implementation. My implementation was to create a lookup table using a hashmap containing a usize and a hashmap of usize and a vector. This hashmap would store the large block that the index was in, the smaller block the index was in, and then the rank at that position of the index (only dependent on the ranks in the small block it was in). Adding this to my build_rank caused me to have to rewrite it a few times until I was able to perfectly loop through and add the rank of each index and keep the space until log n.

It was important to understand the relationship between the run time and the size of the Bitvector as well as the overhead vs. the size of the Bitvector. It was also critical to compare my results to the actual theoretical results. *Graph 1* represents Rank Size Vs. Time with *Table 1* showing the exact data points. According to the theoretical bound this graph should show a straight line that is O(1). My implementation was not able to perfectly satisfy this but it gets quite close by having each time be within a µs of each other.

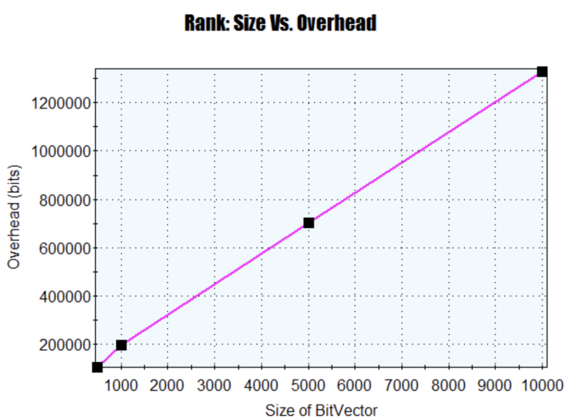


Bitvector size	Time
500	5.318µs
1000	5.394µs
5000	5.49µs
10000	5.694µs

Graph 1: showing the slight change of Run time as the size of the Bitvector grows

Table 1: This shows the values for the points on Graph 1

Graph 2 represents the Rank Size Vs. Overhead with *Table 2* showing the exact points that are displayed on the graph. The space that overhead is supposed to take up as size grows is o(n) which can be seen perfectly by the line made in my graph.



Bitvector size	Overhead()	Number of rank operations
500	106816	8
1000	197760	8
5000	704000	8
10000	1329920	8

Graph 2: showing the change of the Overhead as the size of the bitvector grows

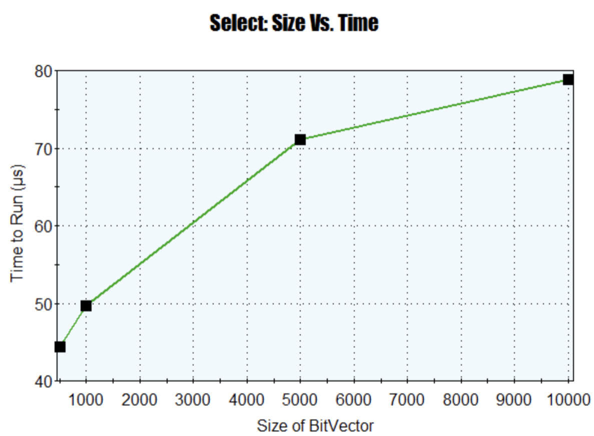
Table 2: This shows the values for the points that are shown in Graph 2

Task 2

To create the select function I focused on creating a function that took up o(n) space while taking O(logn) time. To do this I knew I had to focus on using the rank1 function from task1 while also doing a form of binary search. The first implementation starts with the lowest point being the 0 location in the vector and the highest being the length of the vector. From here the middle is chosen which will be (l + h) / 2. The only issue that occurs here is that we only want to look at the 1's and not any of the 0's in the bitvector. While keeping track of the original m the implementation continues to loop forward until an m = 1 is found. Then it uses that m to compare to the rank that we are looking for. If they are equal this is the index that is returned otherwise it goes right or left depending on if the comparison was larger or smaller. (this is located in the select 1 function)

The most difficult part of this task was making sure that all the binary search queries landed on a 1 instead of a 0. Normal binary search loops through all the elements of the vector but the one created for this implementation only needed to focus on the 1's in the vector. To implement this I still used a normal structure of a binary search but when I reached a middle index that was a 0 I would loop forward until a 1 was located. While looping usually adds a lot to the time complexity this was okay because the loop would not go past the rightmost point in the binary search or the length of the string. Since we were in the middle the maximum length the loop could go would be half of the bitvector, O(n - m).

Similar to with Rank above it is important to understand the difference in the relationship between the size of a Bitvector and the run time and overhead. The typically expected theoretical bound for Select with Time Vs. Size is O(lgn). I was able to achieve this time complexity which can be seen in *Graph 3*. *Table 3* contains the points plots on *Graph 3*.

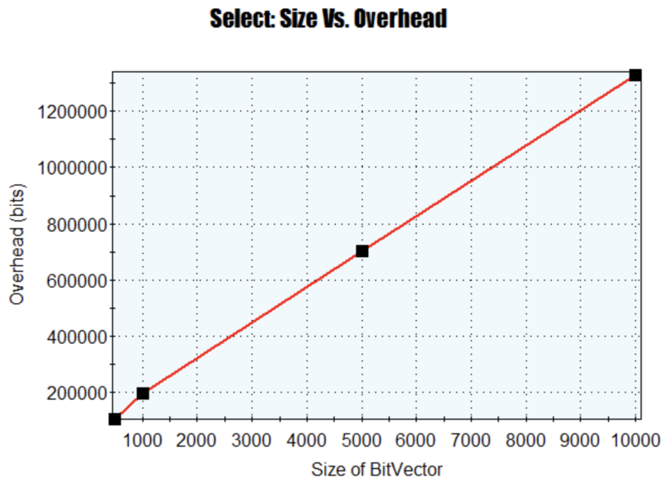


Bitvector size	Time	Number of rank operations
500	44.376µs	8
1000	49.632µs	8
5000	71.105µs	8
10000	78.807µs	8

Graph 3: This displays the relationship between the size of the Bitvector and the Run time of select.

Table 3: This shows the exact values for the the points on Graph 3

Because of the constant nature of the overhead, it would take to build Select and Rank they have the exact same graph for overhead. This graph also follows the theoretical bound for overhead by being o(n). The graph and its exact data points can be seen in *Graph 4* and *Table 4*.



Graph 4: This displays the relationship between size and overhead for Select

Bitvector size	Overhead
500	106816
1000	197760
5000	704000
10000	1329920

Table 4: This displays the exact values for the points on Graph 4

Task 3

This task required creating a struct that had a select structure, bitvector, hashmap of the values, and a boolean that says if it is finalized. These are used throughout the 11 functions that are required to be implemented in this task. Most functions require information that can be pulled directly from the structure, like *size,* which can be found using the length of the bitvector. Other functions require the use of functions from select and rank to help fill in gaps in information. These functions include num_elem_at, get_index_of, get_rank_at, and finalize.

The part of this task that I found to be the most difficult was using functions from both select and rank. While these functions are incredibly useful, Rust can make it hard to access the information in them correctly, with all the different ownership and references required. Particularly because I wrapped everything in options so they could be initialized as an empty structure this meant that a lot of unwrapping had to be done. All the different function calls in one line can tend to make it slightly confusing and when debugging, make it easier to get lost.

There were a lot of different functions that could be tested to do an analysis of the relationships with the size of the sparse vector and sparsity. To make this similar and not get overwhelmed with data I choose 5 different functions from the Sparse class to test: get_index_at, finalize, get_index_of, num_elem_at, and size. To first try to answer the question, how does the overall size of the sparse vector change the speed of the different functions? I make sure to use constant numbers for each function at a different size and use a sparsity of 25%. The results of this experiment can be seen in *Table 5*.

While all of these functions appear to have various changes in run time depending on the size some are more constant than others. Size seems to have a very similar time to build ranging by only 9ns apart. On the other hand, finalize has a drastic increase in the time it takes to complete it as the size of the sparse vector grows.

Functions Used/Size	1000	10000	100000	1000000
get_index_at	395ns	506ns	404ns	595ns
finalize	1.760595ms	12.192888ms	125.687753ms	1.22438629s
get_index_of	181ns	18.997μs	17.287μs	27.03μs
num_elem_at	2.855μs	1.309μs	1.682μs	1.764μs
size	47ns	51ns	43ns	52ns

Table 5: This displays the change in run time when varying the size of the sparse vector through 5 different functions. A constant sparsity of 25% was used throughout.

The next question that was important to test was how sparsity affected the speed of functions. To do this I used constant variables for each function throughout the different sparsities and had a constant sparse vector size of 100000. The results of this experiment can be seen in *Table 6*. One very interesting part is that while changing the size of the sparse vector drastically changed the time it took to produce finalize, it is clear in the graph that sparsity has little to no effect on finalize. It also appears that for all functions, except num_elem_at, the sparsity of 0.01 took the longest to complete.

Functions Used/Sparsity	0.01	0.05	0.15	0.25
get_index_at	498ns	381ns	434ns	409ns
finalize	123.517263ms	123.471749ms	123.707376ms	125.062613ms
get_index_of	96.751μs	28.982μs	20.256μs	19.438μs
num_elem_at	1.399μs	1.671μs	1.653μs	1.506μs
size	61ns	40ns	38ns	38ns

Table 6: This table shows the change in run time when varying the sparsity of the sparse vector and the effect it has on 5 different functions. A constant sparse vector size of 100000 was used to ensure consistency.

Finally, if this sparse vector was implemented with "empty" elements instead of 0's I do not think it would help save a lot of space because the space appears to be consistent no matter what size sparse vector you are trying to create.